Development of Veda, a prototyping tool for distributed algorithms

Claude Jard, Jean-François Monin, Roland Groz

INRIA

UNITÉ DE RECHERCHE
INRIA RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tel.(1) 39 63 55 11

Rapports Techniques

N°87

DEVELOPMENT OF VEDA, A PROTOTYPING TOOL FOR DISTRIBUTED ALGORITHMS

Claude JARD
Jean-Francois MONIN
Roland GROZ

OCTOBRE 1987

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

Development of Véda, a Prototyping Tool for Distributed Algorithms *

Développement de Véda, un outil de prototypage des algorithmes distribués

Claude JARD†,‡,
Jean-François MONIN‡ and Roland GROZ‡

Publication Interne n° 372 - Août 1987 - 42 pages

† :*IRISA, CNRS, Campus de Beaulieu,*
F-35042 Rennes Cedex, FRANCE
email from uucp : ...!seismo!mcvax!inria!irisa!jard

‡ :*Département Evaluation et Validation de Protocoles*
CNET LANNION A, Route de Trégastel, BP40,
F-22301 Lannion Cedex, FRANCE

Key Words :
Simulation, Verification, Distributed Algorithms, Protocols, Software Engineering, Estelle, Prolog.

*will be edited in a special issue of IEEE Transactions on Software Engineering on Computer Communication Systems, November 1987. This work contributed to the French Research Program C^3 on Parallelism and Distributed Computing.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Abstract

We report our experience in developing a simulator, called Véda. Veda is a software tool to support designers in protocol modelling and validation. It is basically oriented towards the rapid prototyping of distributed algorithms, and has been available for more than two years. Algorithms are described using an ISO formal description technique, called Estelle. We first give an external view of Véda, and particularly how one can describe service properties and tracing, using a specific feature of Véda, called observation. Then, the development of Véda and its internal structure is presented, emphasizing the use of Prolog as a software engineering tool. Typical uses of Véda that have been made in the relatively large community of its users are sketched. We conclude with a critical analysis of the main features of Véda and how they may have contributed to its success.

Résumé : Nous rapportons notre expérience dans le développement d'un simulateur, appelé Véda. Véda est un outil logiciel destiné à aider les concepteurs dans la modélisation et la validation d'algorithmes distribués. Il est conçu pour faire du prototypage rapide d'algorithmes distribués et est disponible depuis plus de deux ans. Les algorithmes sont décrits dans un langage formel issu de l'ISO, dénommé Estelle. Nous commençons par donner une vision externe de Véda, et particulièrement comment on peut décrire les propriétés de service et l'affichage de traces, en utilisant une fonction spécifique de Véda, appelée observation. Le développement de Véda et sa structure interne sont ensuite présentés, mettant en relief l'utilisation de Prolog en tant qu'outil de génie logiciel. Les utilisations caractéristiques de Véda qui ont été effectuées dans la communauté relativement large de ses utilisateurs sont évoquées. Nous concluons par une analyse critique des possibilités de Véda et comment elles ont contribuées à son succés.

1 Introduction

In this paper, we report our experience in developing a tool (Véda) that has already been in use for more than two years. It was developed while the three authors were in CNET Lannion. It has been used by at least a dozen laboratories in France. We present the *current* features of this tool. A first overview of Véda is available in [Jard 85b].

Véda is basically oriented towards the rapid prototyping of distributed algorithms and protocols. We consider that these two notions are synonyms : they refer to programs for handling information that is distributed in space.

Véda has also been one of the very first tools based on the Estelle language, an ISO proposed standard for protocol specifications.

1.1 Formal specification for protocols

It is commonplace to say that the decrease in hardware costs and the development of networking facilities have led to a proliferation of protocols and related needs. However, the development of that kind of software has raised specific problems. Conventional software methods and tools were ill-prepared to deal with some of these problems. For instance, specification and design of a software sequential module can be informal up to the point of code writing. Only very large software developed by several teams would require strict definitions of interfaces, and these are only a small part of the whole module. Protocols are very different in that respect: interfacing with other implementations designed by other vendors is the core of protocol definition. This is why a precise - if not formal - specification is of paramount importance. And it is a important to note that problems (such as specification, debugging...) arise even for the simplest protocols. In particular, distribution raises exponential state explosions.

1.2 Rapid prototyping : simulation vs prototype implementation

There is a need -as yet unsatisfied- for rapid prototyping tools and debuggers for distributed systems. Unravelling the possible behaviours of all but

the simplest (alternating-bit-like) protocols requires some kind of *experimentation*. This can be done on a prototype implementation. However, on implementations, rare events (such as disordering of messages on a link) are unlikely to appear. Unfortunately, validation of a protocol precisely requires some investigation of error handling in unhealthy situations.

Simulation in that respect is a much more powerful tool than a prototype implementation. Here is a short list of its advantages :

- Setting up a simulation takes much less time than a prototype implementation (where everything has to be programmed, often in low-level language), and costs a lot less.
- Simulation makes it possible to have a complete control over all parameters and events of the system under scrutiny.
- Simulation can be done in a virtual time. Whereas an implementation is slowed down by delays on e.g. message transfer, this can be represented by an event on a simulation scheduler that runs on a speeded up virtual time scale.
- It is possible to include observation devices of the kind introduced in Véda: almighty daemons that can scrutinize "on-line" all of the behaviour of the distributed system simulated.

In short, simulation makes it possible to *control* and *observe* a distributed system much better than would be possible on an implementation. Some parameters cannot be tampered with on an implementation: *time* in particular. Global time for instance is an abstraction that may not be found in a really distributed system (cf Lamport and prior to that Einstein...); but many service properties are stated with an implicit global time in mind, and simulation makes it easy to produce this global time.

Nevertheless, a prototype implementation is likely to be useful as a complement to a simulation. The reason is that a simulation is only a model of reality. And certain types of error situations (such as cascading failures) may not have been taken into account in a simulation.

1.3 Background of Véda

Véda was designed on the basis of these premises. But before designing a tool dedicated to protocol simulation, we had experimented the basic ideas of simulating protocols on a few significant examples. In particular, other groups in PTT research centers had submitted to us a realistic protocol for mutual exclusion that was able to work in a very tough environment (loss of messages, site failures) and a session-layer protocol. These preliminary investigations had shown that simulation was a valuable approach that could exhibit very complex error situations that had gone unnoticed on existing implementations [Groz 85]. All the simulations had been done using a simulation language called Simone; Simone is based on Hoare's monitors. It turned out that such a tool, dedicated to general purpose simulations was not well suited to our needs (see section 2.2 for a rationale). This is why we built a specific tool, and we based it on Pascal (for the implementation of the simulation).

Thanks to the use of Prolog, we were able to develop Véda in a short time: a first version of the tool was available and distributed one year after the start of the project. Since that time, and to our greatest surprise, Véda has attracted an astounding number of users from PTT, universities and research laboratories. Véda proved able to handle protocols quite complex and of a reasonably large size. It appeared that Véda filled a need that was not yet covered.

1.4 Organization of the paper

Section 2 gives an afterthought specification of Véda. In that matter, we have followed the advice given in [Parnas 86]. Of course, this is an idealized view, but it is a convenient way of abstracting from all sorts of contingencies that arise naturally in the course of setting up a novel design.

In sections 3 to 5, we give an external view of Véda. We first present the whole tool, and then how one can express protocols and service properties. The internal structure of Véda is presented in section 6. Important design trade-offs and choices are discussed. Part 7 names a few typical uses of Véda that have been made in the community of its users.

The last section is devoted to a critical analysis of the main features

of Véda, its strong and weak points. We try and see how they may have contributed to its success.

2 Afterthought specification of Véda

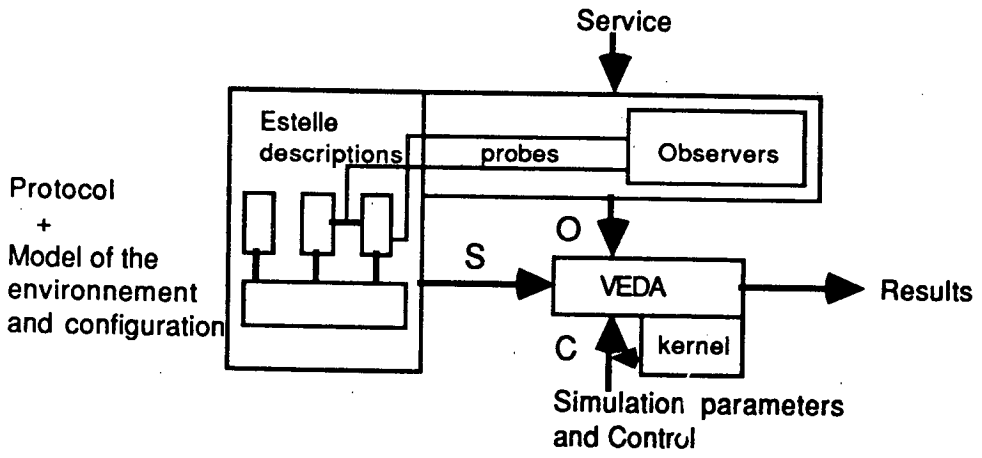
2.1 The protocol design methodology that Véda implements

Véda was designed with the view that the design and validation of a protocol could follow the following steps. First, the designer describes formally the protocol; to that aim, a so-called Formal Description Technique is used. Provided this technique is well-typed, some kind of consistency (at a static level) can be checked at that point by a compiler. In the next step, the simulator already comes in. These preliminary simulations are meant to be short and designed to find out trivial errors (deadlocks for instance), For this task, the designer can use random and interactive (step-by-step) simulation. Fine tuning the protocol comes as a later stage; in that case, it is necessary to have means of scrutinizing all the parts of the protocol behaviour; for instance, tracing of specific messages, or more generally speaking, of all sorts of events (internal states reached etc), is necessary. The last step consists in testing thoroughly the protocol specification. This is the time for long simulation runs. In the current state of Véda, those simulations are done in random mode. Producing huge trace files in that step would be worthless. The method proposed is to verify properties of the system on-line, and at the same time to pick up some measures. All this is done with observer daemons.

Of course, those steps are only a general framework. Véda can even be used for other purposes than protocol validation. Typical uses of Véda are listed in section 7.

The simulation makes use of several entities. First, the system under study (in particular, the protocol considered). Then, the instruments for control and observation. It is important that the protocol specification should not be modified for simulation purposes. It should be kept unpolluted with such things as traces, or error messages. In turn, the system under study can be divided into several entities. The protocol itself should be

Figure 1: External view of Véda



described separately from the model of its environment (underlying layer, and model of user requests from an upper layer for instance). Separate compilation helps in dividing the system in different files.

Figure 1 is a simplified view of the corresponding dataflow in Véda. Véda simulates the execution of the closed system S, under the control of C, with O overseeing this execution (verifying properties and producing traces).

Concerning O and C, several approaches can be considered : they can be compiled (along with the system), or interpreted in the course of the simulation.

Modularity is an important feature of the language used for protocol specifications. In this way, the architecture of the system considered can be represented. However, the more complex the system specification, the more difficult the observation: because the observer is external to the system, it must be able to *name* the objects it observes inside a whole hierarchy of structures.

It is important to note that in Véda we simulate only closed systems. This means that apart from the protocol under study, all its environment (underlying network and user presenting requests for instance) must be modelled somehow. Of course, whereas the protocol must be accurately specified, we can do with a simplified model of its environment. In partic-

ular, if we consider a level 7 protocol in the OSI model, all the underlying network can be represented simply by one single module, a centralized simulation of a distributed network. Thus, the size of the environment can be kept small compared to the protocol specification, even for large applications (see section 7.1).

2.2 Validation-oriented simulation

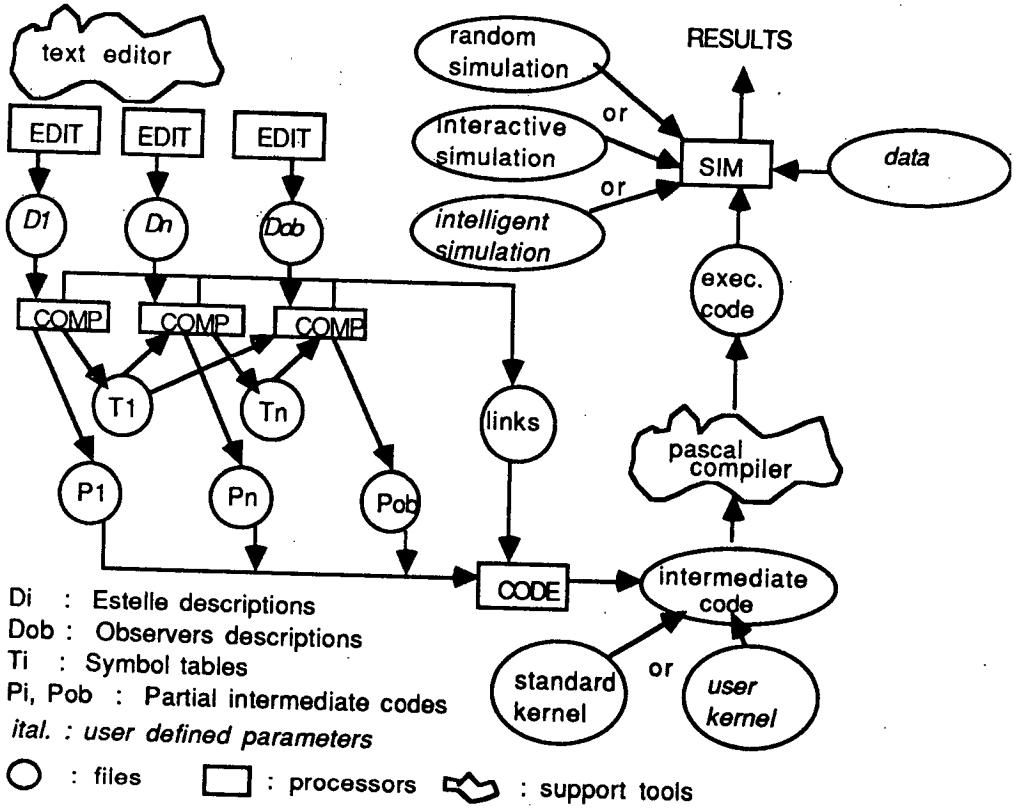
If we try to sum up the functions expected from Véda, and what the implications are on the tool, we see that it must :

- accept formal protocol specification and simulate their execution;
- accept a description of service properties, and provide the means of verifying them; this implies that all events and objects semantically relevant (internal states, transitions, interactions etc) should be accessible;
- provide the means of driving the simulation, especially towards situations than can be interesting (e.g. error-prone); this implies that the scheduling of processes, the time queue, the management of nondeterminism should all be parametrized or even "programmable".

Of course, there is a trade-off between fine control and efficiency of the simulation. The requirements we set imply that our simulator cannot be as efficient as cruder ones, such as Simscript or Simone. Such simulation languages are oriented towards performance evaluation. This is probably why these languages do not offer the access function we need, at least not at the level required for Véda. Note however that the observation constructs required for Véda make it easy to express, among other things, measures of performance. So, if we were to build Véda on a simulation language, we would have to reprogram a scheduler, a time queue and so on on top of those provided by the underlying simulator.This would be a gross underuse of the supporting simulation language, and would result in added-on inefficiency.

As a matter of fact, this was our experience with a pre-prototype experiment done with the Simone simulation language. This is also the reason why we developed our own simulation tool and based it on a general-purpose

Figure 2: Data flow



language (Pascal). It remains however that Véda is inspired from classical simulation languages in that it is optimized for long random simulations.

3 An external description of Véda

3.1 Main functionalities

The overall structure retained for the use of Véda is shown in figure 2.

Some intermediate files show up : every compilation translates a source description (D_i or D_{ob}) into data structures and corresponding access procedures (in P_i), and saves the compilation symbol table in T_i; those tables are used for separate compilation, which also makes use of a "link" file

to manage the tree of dependencies between separately compiled entities. Note that truly separate compilation is performed, not independent compilation: when external identifiers are used, it is clear from which other source description they come, and the compiler performs all semantic checks required.

The "code" command gathers the partial object codes (P_i) and binds them with the simulation kernel to produce executable code. This is link editing, but at a higher level than usual (our object code is pseudo-Pascal, and the executable code is Pascal: so we run on a virtual Pascal machine). The kernel contains all the basic primitives needed for handling processes, interactions, parallelism... A procedure that performs random simulation is included. Also included in the kernel are constants setting up limits (such as the maximum number of processes, 200 by default): however these limits can be modified freely by the users; the only limit is that of the memory available on the host machine. In fact, the whole kernel, far from being a well kept secret of Véda, is at the user's disposal, who can tailor it to his needs, or to the needs of each application.

An interface is provided to drive the simulation (apart form the random mode that is provided in the kernel). When the "sim" command is issued, the user is given a choice between automatic (random) simulation, and an interpreted mode under the control of a Prolog program which can be defined by the user. A default program is the interactive mode in which all the driving is left to the user at the terminal.

3.2 Overview of other commands

Every command of Véda can have arguments and options. Since option names are fully explicit, an abbreviation facility is provided.

Extensive on-line help is available. It covers not only the commands, but also the overall structure of Véda, miscellaneous manuals, the BNF of Estelle and variants used, or news about the features of the last version publicly released.

3.3 Moral

Considering what had been set as a goal for Véda, this tool lives up to its commitments. We must say that some of those functionalities that deal with nonautomatic driving are currently underused, being not user-friendly enough. They can be seen as an open direction, that will be much improved in the commercial version of the tool.

In fact, experience has shown that random simulation is by itself convenient enough to cover most needs [Jard 84]. Its is always possible to make a suitable choice of critical parameters of a model (such as failure rate) so as to force very likely would-be errors in the protocol to show up. But we are still investigating the potentialities of intelligent driving of a simulation.

4 Choosing Estelle

4.1 Estelle among other techniques

Using a suitable language reduces modelling risks : what must be validated is the model, not the modelling. For our needs, the language must include the concepts commonly required for expressing distributed algorithms and architectures. Since a good number of languages include concurrency, we set out to see how they could meet our requirements.

However, we found many languages to be either too general or too restricted. On the general side, we could have considered ADA. But it was beyond our scope and efforts to implement such a language, and it does not offer a specific framework for distributed systems: it is an implementation language for centralized systems, rather than a specification language for distributed systems.

On the other side, a language like CSP, in its 78 version [Hoare 78], is quite popular in the academic world for describing small distributed algorithms. But this language was lacking in syntactic constructs for comfortably undertaking to describe large protocols. It was not sufficiently complete for our needs.

We also turned to methods based on transition systems with a "graphic" base like Petri-nets or SDL [CCITT 84]. Our experience with Petri-nets (which we also use for verification) indicated that protocol modelling with

them was a non-trivial task. And SDL, in 1983 (so the 1980 CCITT version) was incomplete: our observations on other groups within PTT indicated that it was used too informally, with inconsistent specifications as a result. The situation is much brighter in 1986 with tools coming up all around the world.

As to formal calculi such as CCS and Lotos [ISO 86b], they were not well-advanced enough as languages when we started. It is not quite sure whether protocol specifiers are ready now to take on such formal methods.

We finally chose Estelle (or subgroup B language as it was known), with which we were already acquainted: we had based our preliminary experience on the language defined by Gregor v. Bochmann [Bochmann 78]. Although this language includes a lot of syntactic sugar even for simple distributed algorithms, and is quite complex with little orthogonality, it has been tailor-made by protocol experts, and is well accepted by rank and file users thanks to its Pascal base. And the fact that it was in the process of becoming a standard was a keypoint in our choice.

However, we made this choice in 1983. Estelle was not completely defined at that time. So we relied on what was the most up-to-date official document: recommendation X.250 of CCITT. We just extended it with a simplified description of static architectures. So the current language implemented in Véda is known as FDT-E. It is in fact a strict subset of the current Estelle [ISO 86a] (with a few minor discrepancies in the choice of keywords). What FDT-E does not include is dynamic configuration, and associated problems (parent-child relations, activities and processes etc). We are currently updating Véda to conform to the new version of Estelle.

As a final comment, we can say that the choice might have been different in 1986. For instance, Occam [May 83], and to a lesser degree Lotos, have gained in strength and support. They are languages with a clear semantics. However, we feel that, had we made the choice of such a formal language for Véda, the appeal of our tool would have been much less to the community of would-be users, and Véda might well have been a "flop".

4.2 A glance at Estelle

Detailed presentations of Estelle are available in [Linn 85] and [Courtiat 87]. In this paper, we shall only present a small example. This example will be

used throughout this paper to illustrate how Véda works on it.

Figure 3 is a description of a toy resynchronizing protocol between two logical clocks. This is a simplified version of a protocol privately communicated by Gérard Roucairol. It gives a sample Estelle specification.

The basic idea is to resynchronize the logical clocks or timestamps (represented by integer variable H) of two stations. The protocol must ensure that their drift is bounded by a given quantity DELTA (which must be greater than 0). To that aim, each station keeps track of its own time (H) and an upper bound (u) on the time of the other station.

The protocol is based on a "window" principle. The window is delimited by H and u, and its maximal width is DELTA. Each station is given a credit of incrementations equal to its window size.

A station can do one of three things :

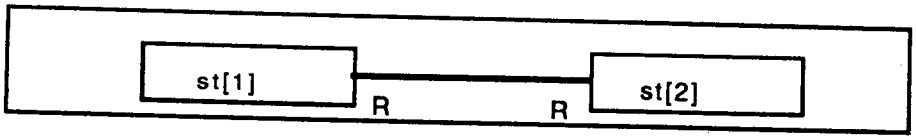
- increment its clock (provided it would not go beyond the authorized upper bound u);
- send its current time to the other station;
- receive a message from the other station, and update u accordingly.

The algorithm appears clearly in the centre of the specification. There are three transitions : two of them are spontaneous (they begin with keyword "delay"), and the last one is triggered by the arrival of an external interaction on gate R (this transition begins with keyword "when"). Preceding the transition part, we find the declaration part, where Pascal constants and types are declared, along with channels and modules (this declares types of external interfaces) and bodies (internal behaviours associated with the external interfaces). The last lines from "modvar" downwards set up an architecture made up of two instances (called `st[1]` and `[2]`) of module type `Station`. They both use the same algorithm (`body CR_simplistic`). These two stations are simply connected together (see figure 4). Although the algorithm can cope with loss and disordering of messages, we have not modelled the behaviour of the underlying communication network in this over-simplified version.

Figure 3: Formal description of a simple protocol

```
specification C_Rouc ;
   const DELTA = ... ;
     d1 = ... ; d2 = ... ;
     inc1 = ... ; inc2 = ... ;
   type logical_time = integer ; one_two = 1..2 ;
   module whole_system ;
   body ws for whole_system ;
     channel Resynchronizing (sender,receiver) ;
          by sender : My_time (x:logical_time) ;
     end ;
     module Station systemprocess ;
          ip R : Resynchronizing(sender) ;
     end ;
     body CR_simplistic for Station ;
       var u,H:logical_time ;
       initialize begin H:=0 ; u:= DELTA end ;
       trans
         delay (inc1,inc2) provided H<u begin H:= H+1 end ;
         delay (d1,d2) begin output R.My_time(H) end ;
       trans
         when R.My_time priority 1
           begin if u<x+DELTA then u:=x+delta end ;
    end ; (* body CR_simplistic *)
  modvar st: array[one_two] of Station ;
  initialize begin
                all i:one_two do init st[i] with CR_resync ;
                connect st[1].R = st[2].R
              end ;
  end ; (* body ws *)
end.
```

Figure 4: Architecture



5 Observation : an interesting feature of Véda

5.1 Overview and rationale

All sorts of methods for expressing protocol properties have been investigated in protocol verification tools. We had first considered the possibility of using some kind of (linear) temporal logic; or finite state machines that would express some regular expression on the ordering of service primitives. We soon found that most formal methods for expressing service properties were :

- limited in expressive power; for instance, with temporal logic or FSM limited to regular languages; and thus, unable to express properties such as "number of messages received=number of messages transmitted" as soon as unbounded quantities appeared;
- restrictive on the events that could be taken into account : for instance, limited to ordering of messages, and considering internal states as not semantically relevant.

Although some of these limitations are well-founded in theory, it turned out that they were unsuited to the practical needs of protocol designers who used Véda. So we were urged to use a more general approach to service properties, and protocol verification altogether. Those interested in a discussion justifying this approach will find arguments in [Groz 86].

The verification in Véda is expressed in a language which is a programming language. The program is called "observer". In fact, there can be one (the usual case) or several observers: this can be used, for instance, to express separately the verification of different parts of the service. Other uses are discussed in section 5.5.

Observers are almighty daemons: they can see every interaction exchanged in the system, and internal states of a module (except in the course of a transition, when the state is not defined). They can halt the simulation while observing. Then, they can compute any recursively enumerable function on what they have observed, with the ease of a programming language. They may be talkative daemons: they can display (or write on a file) their view of what they have observed. So, the power of a programming language is provided for expressing either verification (a computation of some

properties) or a sophisticated trace (displaying only well-chosen high-level events).

The programming language retained for the first version of Véda was simply a variant of Estelle. This choice was motivated by several reasons :

- the state transition model is well suited to observation (cf infra);
- referring to Estelle objects is made easier by using syntactic constructs of Estelle;
- we had a ready compiler for this language;
- users of Véda do not have to learn and master a completely new language.

There is however a drawback in using Estelle: with this model, the observation itself is inherently non-deterministic. This is more often a nuisance than a benefit, because the intuitive view of verification or tracing is rather deterministic. Fortunately, this nondeterminism is of little consequence to the normal use of Véda because observers are usually written in such a way that they are either deterministic or insensitive to the order in which the transitions are written.

There is a major problem to be solved when using observers external to the system observed. We have already justified this choice by saying that it keeps the specification of the system unpolluted with simulation or observation-related features. However, since observers are supposed to have access to all objects within the system, they must be able to *designate* them. Objects of interest in Estelle are interactions and process variables (including major states). The problem lies in the nested architecture that Estelle enforces. Several object instances of the same type will have the same name. And worse, different types may be associated with the same identifier, provided they lie in different syntactic scopes. The basic idea is that of using a dotted notation (just as for Pascal records). However this could be tiresome in the case of deeply nested structures. So a shorthand is provided, with the notion of probes (see figure 7 for a graphic illustration of the notion of probe). An observer has a certain number of probes, each one having a name. An observer can observe only through its probes. And

each probe is set (once and for all) on one object of the system. So the probe has two functions; it serves as :

- a shorthand for referring from within the observer to the object observed;
- a filter: the observer is concerned solely with the objects on which it has explicitely set a probe.

In this way, the description of the observation is divided into two parts. First, a description of probe types, probe instances, and their connections. Second, the observation computation (verification and tracing...) which is expressed solely in terms of probes, with no explicit reference to the system.

5.2 Expressing verification with observers

In Estelle, the evolution of a module is expressed in terms of internal states and interactions. Internal states stretch over time, at least from one transition to the next. Interactions are seen only punctually: when output or input. This duality in nature makes it all the more difficult to have a uniform view of what a property, bearing on both aspects, is. Typically, for states, one may be interested in invariants. Taking into account the transition-based semantics, the observer can specify that an invariant must hold by performing the following computation after every significant transition in the system :

as soon as not [invariant] $\longrightarrow$ error(message for instance)

When an interaction must be observed, we can trigger some action in the observer :

[interaction input/output] $\longrightarrow$ action (for instance, trace it or remember this occurrence).

In that way, a unified view consists in saying that the observer is a set of predicate/action couples :

on [event] $\longrightarrow$ action

Figure 5: An observer

```
(* P is an array of two probes *)
 body obsr for obs
   trans provided abs (p[1].H - p[2].H) > DELTA
          begin writeln ('error') end ;
```

where [event] can bear on internal states and those interactions that are present (occurring) when the observer does its observation. As can be seen, we come naturally to a state-transition model for observers. In addition, the definition of probes necessarily makes use of Estelle constructs. This is the main reason why we were led to use an Estelle-based language for observers.

Let us have a new look at the example pictured on figure 3. This protocol ensures that the drift between the logical clocks H of the two stations is bounded by DELTA. Figure 5 expresses this property (bounded drift), in the observation language of Véda; there remains only to be seen how Probes p1 and p2 are expressed. Note that the observer would remain the same, even for another resynchronizing protocol (more realistic for instance), provided it ensures a bounded drift DELTA. This is another benefit of the probe concept.

5.3 Probes

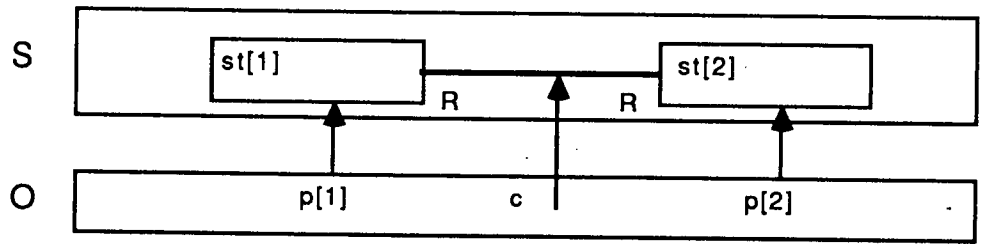
Figure 6 is a complete description of the observation and configuration parts. This seems complex, but it is due to the fact that the example is outrageously simplistic. With a much more complex system observed, the observation part would be scarcely longer than it is here.

As can be seen, the usual two-step definition (type/instance) is expressed in the manner of Estelle. First, probes are *typed:* in that part, the user defines which *syntactic type* the probe will access. This is done in the module declaration of the observer (`module obs observe [probes-type-declaration]`). Then probes are *connected* (in the initialize part of the whole simulation architecture). This defines which *instance* in the system

Figure 6: Formal description of the observation part

```
simulation O_C_R ;
  import C_Rouc ;
      (* uses a separately compiled system    *)
  observe (* introduces the observation part *)
  module obs observe  (* typing of probes     *)
           module P : array [one_two] of ws.CR_resync
           channel c: ws.Resynchronizing ;
  end ;
  body obsr for obs ; (* service property     *)
   trans provided abs (p[1].H - p[2].H) > DELTA
           begin writeln ('error') end ;
  end ;
 modvar S: whole_system ; O: obs ;
 initialize begin  (* setting up a configuration *)
          init S with ws ; init O with obsr ;
          all i:one_two do observe O.p[i] : (S,ws).st[i] ;
          observe O.c  : (S,ws).st[1].R
        end ;
end.
```

Figure 7: Graphic illustration of probes



the probe will observe.

In the example above, we define a probe p that is meant to observe module instances of type CR_resync. However, scoping rules imply that CR_resync is not visible when the observer is declared : CR_resync is a body which is declared within body ws of specification Carvalho_Roucairol. Importing Carvalho_Roucairol gives access to objects declared at the top level of this specification (e.g. ws). To denote CR_resync, we prefix it by the path of syntactic constructs that give access to it. In this simple example, the path is only of length one, so we get :

```
ws.CR_resync
```

After that, when the architecture is defined, we declare only one observer (instance) O. So there are two probe instances O.p[1] and O.p[2] that must be set to observe modules. Again, we must give a path, but of instances this time. So we would expect, for this path, S.st[i].

However, st[i] could be ambiguous. The reason is that another body, say ws_bis, could have been declared for whole_system; and ws_bis could contain an

```
st :  array[0..3] of Station ;
```

Since the architecture is set up at runtime, the compiler cannot know which st is referred to. This is why we force to write a path of couples : (instance,init-body).

In this way, many checks can be performed at compile-time. And the access is precompiled, so that observation is quite efficient at runtime.

5.4 Tracing

Véda offers a set of predefined procedures for tracing interactions (messages) comfortably enough. With these procedures, it is possible to display messages and their parameters in high-level terms. Even if a parameter is a "record of arrays of sets of enumerated types..." the content would be listed in high-level terms (braces for sets and so on). The display, as shown on figure 8 includes the virtual date, an identification of the interaction point where the interaction took place, and a high-level description of the

Figure 8: Example of tracing

```
body trace for obs ;
  trans
     begin writeln('h1:',p[1].H,'; h2:',p[2].H) end ;
  trans
     begin exrepprobesmess end ;
end ;
  ..............................................
     yields the following trace :
  ..............................................
    h1:    0 ; h2:    0
    h1:    0 ; h2:    1
      6.653 : S.st[1].R --> My_time (0)
    h1:    1 ; h2:    2
     11.054 : S.st[1].R <-- My_time (2)
    h1:    1 ; h2:    2
     13.896 : S.st[1].R <-- My_time (2)
    h1:    1 ; h2:    3
```

message, its direction (input/output) and its contents. This is a default display, but it can be tailored by the user.

The trace shown on figure 8 can be obtained by having the two transitions mentioned in the observer. These two transitions have no condition (implicit "provided true"), so the trace is always enabled. The second simply calls procedure exrepprobesmess that traces all interactions that pass through the points observed.

In the first steps of specification debugging, it might be tedious to have to write an observer with probes on all interaction points just to trace what happens. This is why Véda offers, as an option in the "code" command, a facility to include automatic tracing of all interactions (an equivalent of the exrepprobesmess procedure); this facility is implemented in the kernel, so it is available for the trace of systems that are simulated with no observation at all.

5.5 Other uses

Observers have been used for performance measurement: it is easy to collect data on the progress of the system and produce a statistical analysis at the end of the simulation. However, implementing data collection in this way is not efficient. And performance evaluation does not make much sense on a non-deterministic specification. So we are currently investigating which better-suited methods could be implemented in Véda.

It is also possible to have several observers running in parallel, and communicating with each other. This has been used for experimenting distributed testing schemes [Dssouli 86].

Observers could also be used actively, interacting with the system. Although this facility was foreseen in the design, it has not been implemented in Véda, because the semantics of the evolution of the system would be unclear.

6 The implementation of Véda

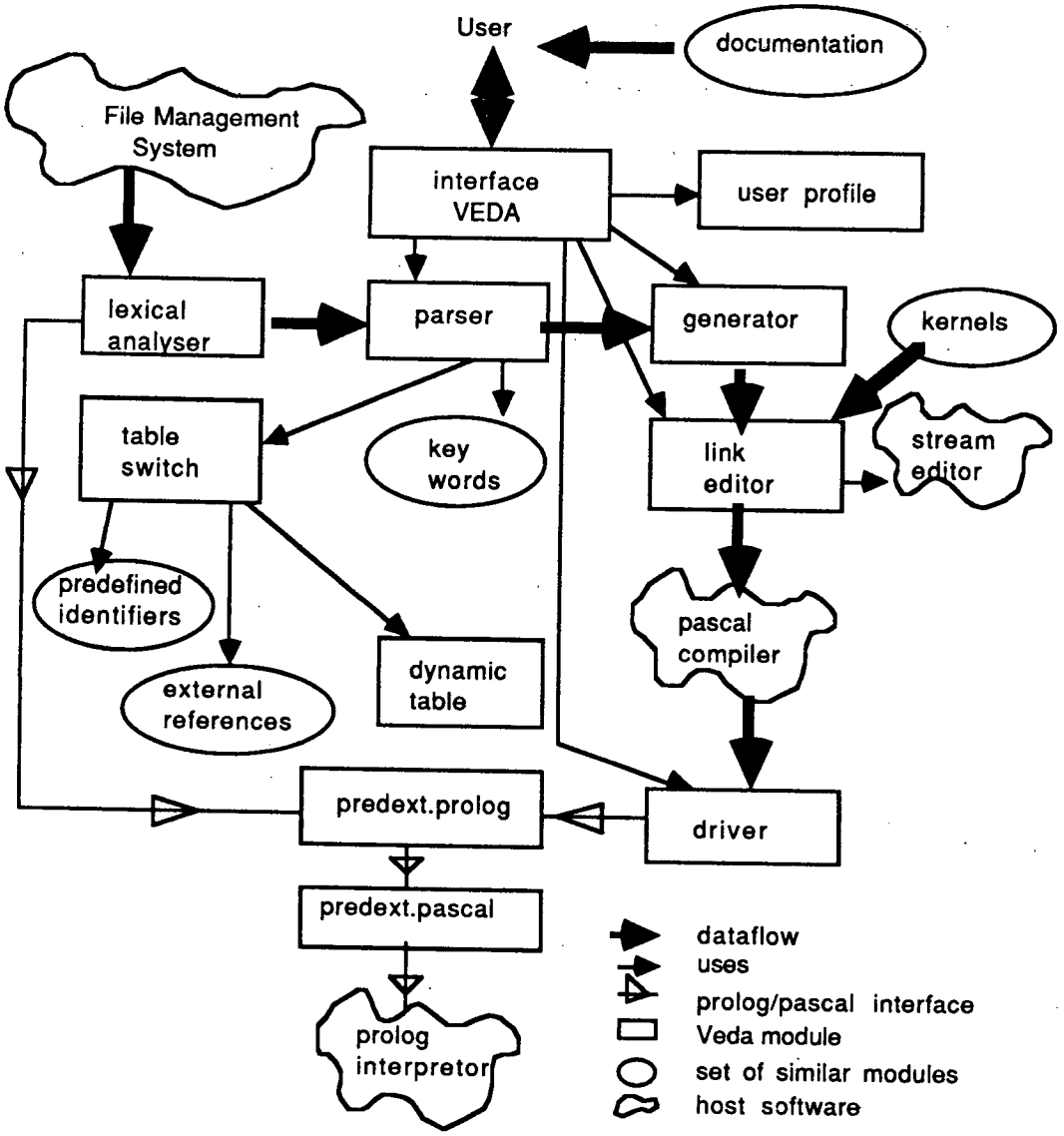
6.1 Overall structure

The functions presented in section 3 are implemented by the software modules represented in figure 9. Véda is made up of a number of boxes driven by the central box called "interface_VEDA". The dataflow , represented by full arrows, may be compared with the one on figure 2.

"Interface_VEDA" is an interpreter of commands from the user. It contains a command analyser (including the abbreviation handler), and a command manager that invokes the procedures corresponding to each command. The profile proper to each user, in particular the abbreviations he has defined, are kept in the "user_profile" module.

Compilation gets the lion's share in the software involved. Compiling is done in two passes. The first pass performs lexical, syntactic (context-free) and semantic (viz. contextual) analysis, and produces an intermediate code. It also produces, as a by-product, the symbol table (for use in separate compilation). If no error has been detected, a second pass produces object code from the intermediate code. The first pass is associated with the "parser" box and the second with the "generator" box.

Figure 9: Internal structure of Véda



The first pass calls the lexical analyser, and uses symbol tables: one for predefined identifiers (with a choice between French and English, as for keywords), and one for each separately compiled unit used ("external references"). The "dynamic table" box contains the symbol table for the unit currently being compiled. All these tables are kept in different Prolog modules (one per table), and accessed through the "switch" box.

All the boxes correspond to Prolog programs, except the lexical analyser which is written in Pascal. We discuss this choice of implementation languages in sections 6.2 and 6.3; briefly, we can say that in this way, the time required for a compilation is quite tolerable, even though it is slower than a fully compiled compiler.

The "link-editor" box corresponds to the command named "code" in Véda. This command assembles the object code produced and the simulation kernel to produce an executable code. In fact, the object code is made of bits of declarations and procedures which are mixed with those of the kernel (see section 6.2.1). The task of the link-editor is first to find out (from the "link" file that keeps a record of the tree of dependencies between separately compiled units) which object codes are to be fetched; then the link editor processes the tokens of the object codes into unique identifiers, avoiding conflicts (clashes between identifier names). The assembly is done with the help of a stream editor; so the only thing that the link editor does is to build a command file for this stream editor. This is done again by a Prolog program. Lastly, the "executable code" produced, which, for us, means Pascal code (since we run on a virtual Pascal machine), is compiled on the host Pascal compiler.

Now this code can be executed to represent a simulation of the distributed system. This is done by a program called the "driver", which corresponds to the "sim" command of Véda. The driver implements some simulation strategies, by calling suitable entry-points in the executable code. Exemples of extreme predefined strategies are: fully random, or fully interactive.

There remain two boxes to be explained: "predext.prolog" and "predext.pascal". Those result from the choice of Prolog as main implementation language of Véda. In these two programs, the interface between Prolog and Pascal is defined. In particular, the dividing line between Prolog and Pascal passes through the lexical analyser and the driver (both programs

have a part written in Prolog and a part written in Pascal). This interface described how the so-called "external predicates" of Prolog (hence the name "predext") are implemented as Pascal procedures. As an addition, "predext.Prolog" contains the definition of all built-in and general purpose Prolog predicates.

6.2 Implementing and observing an Estelle simulation

6.2.1 Overall structure of the target code

An obsession runs throughout the structure of the object code: to keep full control of the finest elements that could be of interest in the simulation.

The object code is made of two parts, of a different kind: a fixed part (or "system" part, corresponding roughly to the kernel), and a variable part which is defined by the protocol. The system part embodies the management of parallelism, of virtual time, of interactions and the use of memory. The protocol part embodies, for instance, the translation of Pascal types, procedures and functions declared in the specification.

It is good practice to bring the time spent in the system part to a minimum. But for a validation-oriented simulator, the protocol part must be interrupted whenever it is in an observable state, and whenever a driving decision must be made. The definition of these interruption points depends on the semantics of the language simulated. In the case of Estelle, the action part of a transition is considered atomic: while executing, it is insensitive to all external events and its state is undefined. Thus, we will consider transitions to be uninterruptible.

Coming to process management, we see that for each process (we use the word in its general sense, not the specific meaning of Estelle) there is a cycle within each process of determining enabled transitions and choosing (under guidance from the driver) to fire one of them (if any). Then, the parallelism between processes must be simulated. This is a crucial point: running on a sequential machine, a serialization (interleaving) of process actions is unescapable. This introduces a simulation-dependent total order, where only the partial order of causality is semantically relevant. To circumvent this problem, we must be cautious when determining "enabled" transitions.

The independence of process choices outside their interactions must be kept. Consider for instance the protocol described on figure 3. Suppose that only the first transition (incrementing h) is enabled for one process, while only the second (output `My_time`) is enabled in the other process. In that case, the possible simultaneousness of both transitions (transition 1 in process 1 and transition 2 in process 2) should be included in the simulation, even if the serialized version interleaves process 1 after process 2. Note that outputting `My_time` in process 2 disables transition 1 of process 1 because of the priority rule (priority to the input). This means that the enabled transitions of process 1 are not reassessed after executing process 2.

With the (disputable) simplification that transitions can be considered to take no time, the following simulating scheme can be adopted :

loop

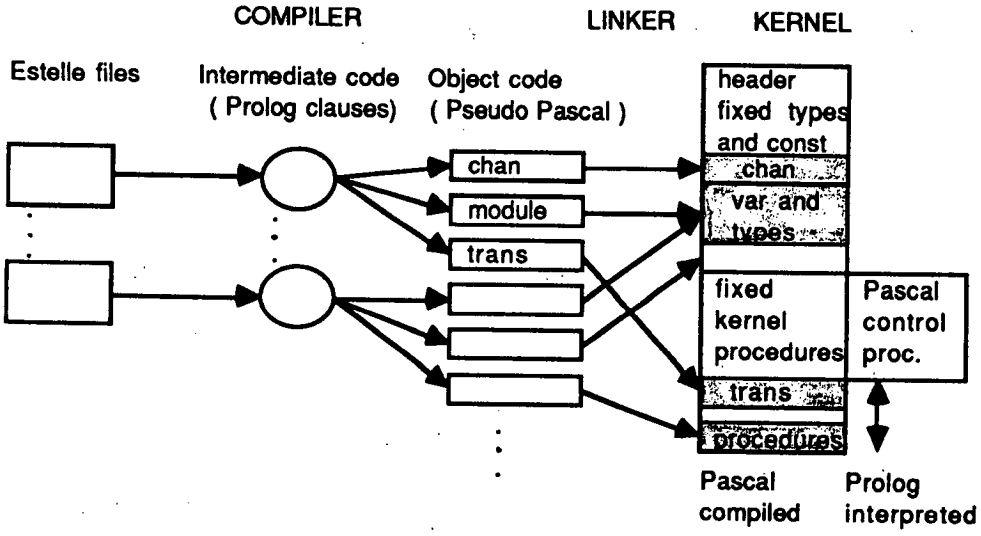
- select (driver's choice) a subset of processes (= those considered as running for that slot of time)
- compute enabling conditions for all transitions of these processes
- select (driver) one (if any) enabled transition per process
- execute the selected transitions
- (observe: this is where observers come in; the interleaving of the processes is not semantically relevant to them).

The default driver strategy is random choice (which is a very peculiar notion of strategy!).

The kernel is structured in three parts :

- General purpose routines: pseudo-random functions, of which three independent sets are provided (for the simulation, for the observation, and for the user convenience); warnings and errors...
- Basic primitives, either data structures or procedures; this includes : configuration ("init" etc), communication and time ("delays"); also primitives for observers only : traces, and the exit procedure that aborts the simulation. The code generated from source descriptions use these basic primitives.

Figure 10: Target code generation



- Higher-level primitives for use by the driver : list of active processes, list of enabled transitions, executing a transition, and determining the possible next events in the time queue; there are also corresponding primitives for observers. Those higher-level primitives are accessible both as Pascal procedures and Prolog predicates.

The code generated from the source descriptions is made of the following structures :

- Type definitions for the data structures representing modules (and the hierarchy of instances), process contexts, and interactions.
- Trace procedures for each user-defined Pascal type that is used in parameters of interactions.
- Configuration procedures, corresponding to an "init", for each body type. A root configuration procedure is also generated.
- A procedure for each process (i.e. a body with a trans part) type definition; inside this procedure, two sub-procedures are defined for

each transition: one for the enabling condition, and the other for the execution.

Of course, this is only a very brief outline of the Pascal structures used to simulate Véda. Those interested in a more detailed presentation of the translation of Estelle constructs, and how the scheduler and time queue work will find it in [Jard 85a].

Generation of the target Pascal program from the Estelle descriptions (variable parts) and from the kernel (fixed parts) is illustrated in figure 10.

Figure 11 shows the different steps in the transformation of an Estelle channel (extracted from the synchronization protocol of figure 3) into Pascal code.

6.2.2 Moral

To what extent did we benefit from using Pascal as target code ? We were able to avoid interpretation or compilation of most Pascal constructs by a careful division between "system" and "protocol" part, and playing on the atomicity of Estelle transitions. From that point of view, the use of the Pascal compiler is fruitful.

On the other hand, it turns out that it is impossible to avoid syntactic and even semantic analysis of purely Pascal parts within Estelle. Far from being a juxtaposition of Pascal and supplementary constructs, Estelle is pervaded with Pascal.

This implies a redundancy in parsing (Véda + Pascal compiler), but Pascal errors in the Estelle source are signalled early (in a much more explicit way than they would be in generated code).

The main problem comes from Pascal runtime errors. In that case, the user of Véda must have a minimal knowledge of the translation from Estelle to Pascal. Although all identifiers from the source are renamed, each identifier in the object code is tagged with a comment containing its original name. This kind of simple help has been found satisfactory enough by users of Véda, taking into account the "prototype" nature of Veda.

For a polished version of Véda, this drawback could be overcome (we could interface with the Pascal runtime or a debugger, or program checks on top of Pascal).

Figure 11: Code generation for an Estelle channel

estelle source text

```
channel Resynchronizing (sender,receiver) ;
     by sender : My_time (x:logical_time) ;
end ;
```

prolog intermediate code

```
mesg(
  id((C_Rouc.4).My_time),
   id((C_Rouc.5).x, id((C_Rouc.6).logical_time)).Nil
  ).
```

partial code

```
, %IDC_Rouc%o4o1{My_time}
 %IDC_Rouc%o4o1{My_time} :
   (  %IDC_Rouc%o5o1{x} : %IDC_Rouc%o6o0{logical_time}
   ) ;
```

pascal code

```
typeinteraction =
   (sivide0
    , o0o4o1{My_time}
   ) ;
typemessage =
    record case sorte: typeinteraction of
        sivide0 : (b:boolean) ;
        o0o4o1{My_time} :
          ( o0o5o1{x} : o0o6o0{logical_time} ;
          ) ;
     end ;
```

6.3 On the choice of Prolog for implementing Véda

The compiler is the largest piece of software in Véda. We put our stakes and hopes on Prolog for it, and on having a uniform framework for writing the whole system (viz. both Véda and the software environment to write Véda).

6.3.1 Compiler

Although compiler-writing in Prolog has been envisioned since 1975 [Colmerauer 75], there was no real-size successful experience when we started. Fortunately :

- Véda compiles descriptions of more than two thousand lines, which is much more than necessary, thanks to separate compilation. Systems consisting of thousands of lines of Estelle have been compiled (see section 7). Remember that we are compiling a specification language, not an implementation language: a thousand line long specification is already a big one.

- The speed of compilation is certainly slow, but tolerable: several source lines per second. The compiler runs on a Prolog interpreter, and it cannot compete with a compiled program. Crucial phases had to be carefully optimized: the management of the symbol table (see [Monin 84]), and lexical analysis. When we switched from lexical analysis in Prolog to a Pascal-written predicate, the speed of context-free compiling was increased tenfold!

- Prolog has proved a very handy language for managing software upgrades. Separate compilation, compiling observers and type-tracing routines have all been introduced with little or no change to the existing compiler. And fixes for bugs in the compiler were very small.

6.3.2 Prolog as a software engineering tool

Managing a software like Véda requires adequate programming methods. One principle which has been used for Véda is: "for each problem, choose the method (language for instance) that fits best". Since Prolog is very

good at language transformations, this approach was easily supported in Véda. For various problems, we designed an adequate formal language for specifying a solution, and built the adequate translator in Prolog. This approach is close to metaprogramming as defined by [Levy 86]. In Véda, we used this approach mainly for writing the parser and the object code generator. Meta-tools used for Véda are shown on figure 12. Detailed analysis of the figure is out of scope of the paper.

An example of meta-pascal-specification of code corresponding to figure 11 is shown in figure 13.

6.3.3 Which Prolog ?

We used Prolog-CNET [Barberye 83] better known as Prolog/P in its commercial version. Two prominent features of this Prolog have proved essential for Véda :

- Modularity. It makes a big Prolog software like Véda more manageable. And it is very useful for managing symbol tables (it avoids automatically clashes between identifiers).
- It is written in and interfaced with Pascal. This means that the public part of the Prolog interpreter (predext.pascal) can be extended with Pascal procedures, which can be called as predicates from Prolog. Of course a compiled Pascal procedure is much more efficient than an interpreted ordinary Prolog predicate. This is what made possible the dramatic increase in parsing performances. But it is also most useful for interfacing the Prolog driver with the Pascal simulation code (cf the primitives described in section 6.2.1). The Pascal simulation primitives are declared as externally compiled predicates. However, this relies on dynamic linking, which is not available on Unix. So this feature is implemented only on Multics.

6.3.4 Moral

Véda proved that a real-size, comfortable software tool can be written in Prolog. It should also be said that the commercial version of Véda will also

Figure 12: Meta-tools used during the development

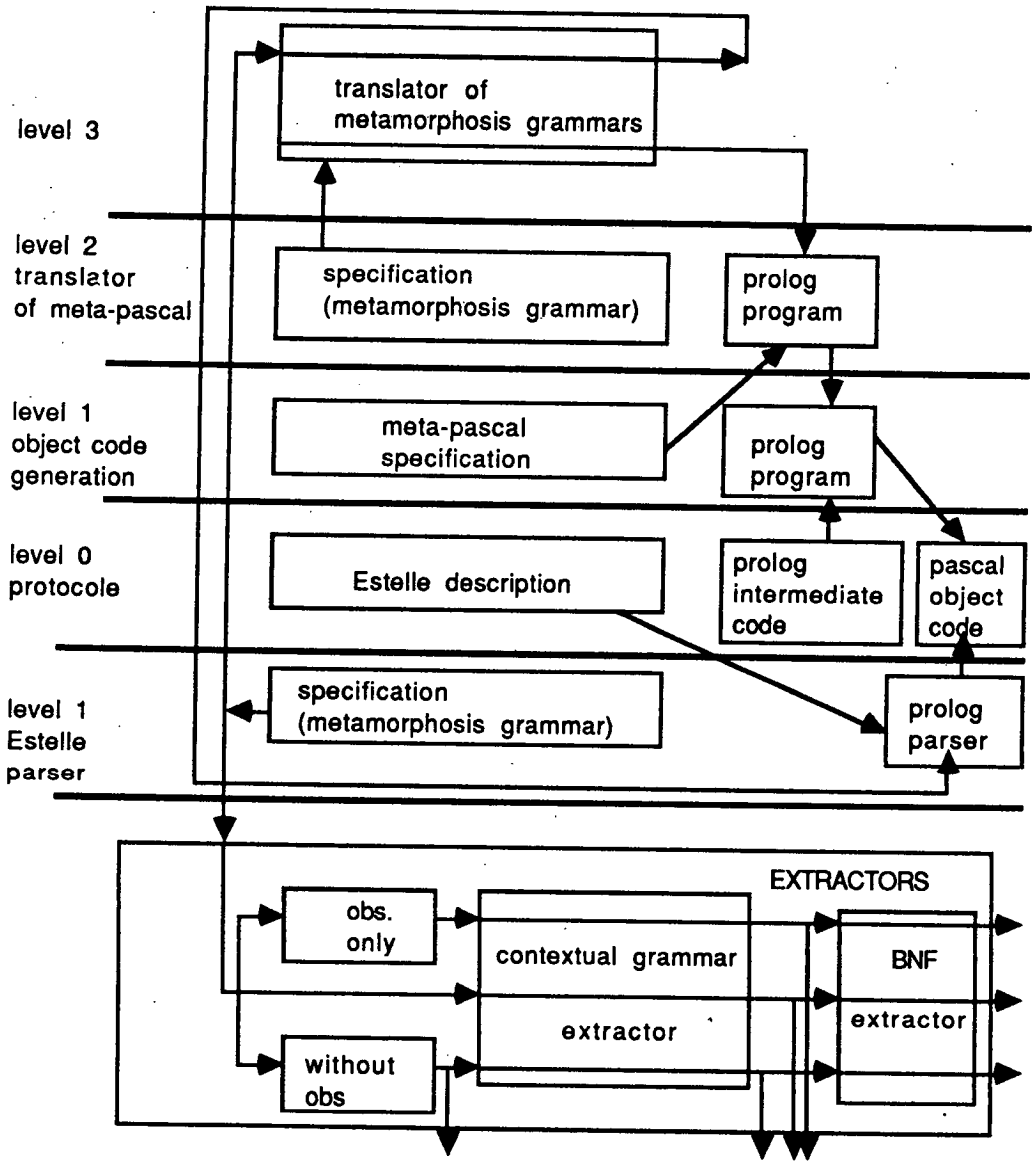


Figure 13: Rewriting rules for interactions

```
+enum_interact(*lmess) ->
     typeinteraction =
         ( sivide0
           +noms_de_messages(*lmess)
         ) ;
+noms_de_messages(mesg(*i,*p).*l) ->
           , +rep_id(*i)
           +noms_de_messages(*l)
+noms_de_messages(Nil).
+struct_interact(*lmess) ->
     typemessage =
         record case sorte: typeinteraction of
            sivide0 : (b:boolean);
            +messages_types(*lmess)
         end;
```

run in Prolog : Prolog can support more than prototypes, it has reached the product level.

7 Applications and distribution of Véda

Véda has gained a large number of users within our PTT research centre (CNET), in french universities and public research laboratories.

In the PTT, Véda has been used for ISDN specifications (we detail that below), radiotelephone protocols, switching designs, satellite protocols, remote maintenance... It has also been used by french manufacturers (Alcatel-Thomson,Bull) on punctual cooperation with CNET.

In our opinion, Véda was certainly not the ideal tool for many of the applications mentioned. But it was there, readily available and easy to use. And only a subset of its functions were needed in many cases. Just to show how the use can depart from the original intentions of the designers of the tool, we give here a selection of typical applications.

7.1 ISDN application

This was an internal study done in CNET (its code name was ARMOR). The basic task was that of defining a functional specification of some parts of an ISDN switch. Véda was used to model the functions involved in the X.213 protocol. The model involved 3600 lines of Estelle for X.213 itself, and 940 lines to model an environment (simulating calls etc). All this was divided into 19 source files. There were 13 module types involved, and 74 different types of interactions. The Pascal code produced for the simulation corresponded to approximately 15000 lines of Pascal (in "normal" style).

In this application, simulation was not the main function required. The need was for a formal specification technique supported by a compiler, with a good degree of modularity. The target was to produce coherent specifications, at a static level. However, simulation came as a plus: it provided a kind of prototype of the functional specification. The evaluation of this example pointed out the strong and weak points of Estelle for functional specification of such systems :

- Strong points: modularity (and hierarchical substructuring), complete definition of interfaces, executability of the specification.
- Weak points: no support of abstract data types; no integration of database concepts.

7.2 Switching

Switching is an essential part of the telecommunications business. Véda was used for validating some protocols internal to components of a large digital switch. There was a protocol at the transport level: the connection part was validated with a Petri-Net tool, the data transfer part with Véda. Other protocols were validated: an election protocol, and a protocol involved in fault-diagnosis.

In that case, the use of Véda followed the ideal pattern described in section 2.1. All the functions were "rightly" used, and the goal was protocol validation.

7.3 Véda as a teaching tool

Véda turned out to be a useful tool for supporting post-graduate courses on distributed systems. It provides students with a possibility of experimenting distributed algorithms, and grasping the difficulty of that kind of design. Véda has been popular in this field because :

- it makes quasi immediate experimentation possible, with no distributed hardware needed;
- all students are familiar with Pascal, whence Estelle (in our restricted version at least) comes easily;
- Véda is user-friendly: extensive on-line documentation, explicit (and juicy!) error messages in French;
- Véda is robust.
- One drawback however: since most of Véda is interpreted (on a Prolog interpreter), its use is expensive for student accounts!

In the case of exercise solving sessions, the use of Véda includes editing, compiling, and simulating but only for debugging purposes (or showing a token behaviour). Intensive validation is only used by PhD students working on larger protocols.

7.4 A failure

To be honest, we must report that in one case, Véda proved unsuited to the needs of protocol designers. Some people in CNET had designed a protocol for collecting maintenance data on remote stations. They were interested in validating their design. But we were not able to offer something interesting to them. This protocol was a low-level one: astride on layers 1 and 2 of the OSI model. The duration of transmission, related to the length of the physical representation of interaction (number of bits required), was important because the protocol involved many synchronizations on these durations. Estelle was ill-suited to represent these low-level constraints. And they were the main feature to be validated.

7.5 Moral

Véda met the needs of many more people than was expected. And the use of a tool can differ to a large extent from what its designers intended. Véda has also proved able to handle reasonably large applications; however, it is best suited for the validation of small protocols, or such distributed algorithms as are published currently in the literature.

8 Attempting to analyse the success of Véda

It is quite usual that people working on research fields like protocol validation should develop tools for their own needs. And Véda fits exactly in that pattern. However, many such tools never go beyond the group that developed them. We thought it interesting to try and see why Véda experienced a success far beyond what we, its designers, ever expected.

8.1 The needs expressed by users

Véda was designed to meet our need in 1983: we were a group devoted to validating protocol specifications defined by other groups in CNET. For this activity, we needed a simulator of specifications. But we were soon pressed with other demands from users. Two main factors have influenced the evolution of Véda.

Firstly, a group specialized in protocol verification techniques soon finds out that few protocol designers are really interested in verification oracles. If no error is found, verification is felt as a luxury. If errors are found, they are either considered to be not significant for the real application, or simply bugs that had passed overseen and can easily (?) be fixed in the next version. However, all protocol designers are interested in a simulation tool with which they can draft a design, experiment and modify it as they like. This is an important lesson we learned from Véda. With it, we were able to pass the burden of preliminary protocol verification from protocol verifiers to protocol designers.

Secondly, it soon appeared that the functions we required (as described in sections 2 and 3) could suit the needs of other people. In particular, Véda could be helpful for :

- designing structured architectures (with Estelle);
- debugging distributed algorithms (esp. with traces and step-by-step simulation);
- teaching distributed systems; and let students have a feel of it.

It has been possible for Véda to evolve accordingly. To that end, we strengthened the presentation part of the tool. This includes facilities provided in the tool to make it convivial (user-friendly) enough. But it also means: providing adequate manuals, setting up training structures etc.

8.2 Corresponding features in Véda

8.2.1 Simulation vs formal proof

Simulation is better received than more formal methods. It is nearer to the culture of the average computer scientist. This is also true for Estelle. And formal proof can play its role best when the design is clear enough. In the current state of the art, it is less helpful when the specification is still vague and incomplete.

8.2.2 Estelle

Learning a new language requires some time and practice. Estelle has the advantage of being based on a widely known language (Pascal) and a would-be standard : this can justify the investment. However, Estelle (like many programming languages) does not permit currently a sound mathematical modelling of parallel computations. Although semantics precision in the area of validation is of paramount importance.

8.2.3 A reasonable quality of the tool

Experimental softwares are often quite limited, user-unfriendly, and sometimes ridden with bugs (or do not recover from unspecified errors). In that case, the software has no chance to be used outside a small group of designers and students.

Such softwares are intended to prove some sort of feasibility. Often, not all the problems are envisioned from the beginning, and some appear unexpectedly in the course of implementation. Fortunately, we avoided that kind of problem for Véda. The use of the "throw-away prototype practice" has probably played a role. Most problems appeared early enough, and could be solved without modifying the general design.

When major extensions were made, they were added to an existing base that was already usable and widely used. This is the case for separate compilation, and observation, which could fit in the original design (they had stood in the back of our minds since the beginning of the project).

8.2.4 Development and distribution on Multics

The operating system used for developing Véda, Multics, has contributed to its success. Multics provides a comfortable environment for software development. We had a good Pascal compiler. Above all, most major research centres in France had a mainframe running under Multics. And they are interconnected with medium-speed, reliable links that permit file transfers and a nationwide uniform mail-system. This enabled us to distribute rapidly Véda all over France.

8.2.5 On-line documentation

On-line documentation provides a detailed user's manual of Véda, and more than that: it also presents other features of Véda that may be of interest (how Véda can be installed on new machines, examples of its use, highlights of its internal structure, news, and even a users forum on some mainframes). This made it possible for plenty of people to use Véda, without their ever getting in touch with us, or getting written documentation; this we could know thanks to a "spy" included in the software to record all users. One weak point however: the tutorial on Estelle (FDT-E) is not available on-line; since we had developed Véda for ASCII terminals, we could not include graphics in on-line help.

8.2.6 Using the vernacular

It turned out that this point is more important than is often deemed. The whole system is in French: user interface, Estelle keywords, on-line help. This is quite helpful for many users (esp. students), and appreciated. And, apart from documentation, providing a multi-lingual interface is quite simple and inexpensive. So why not do it ?

8.2.7 Véda: an open software ?

"Open" is used to mean that a software can be tailored by a user to his needs, or easily extended by suppliers to include new functions. Véda goes some way along this line.

For the user, there is the possibility of modifying the simulation kernel used. In this way, Véda does not set any other limits than those set by the underlying system or Pascal compiler. Code generation, however, is fixed in the 1986 version, which is a strong limit. Observers are also an implementation of this openness concept. There is no limitation on the properties one can express.

Concerning the addition of new functions to Véda, some of them have been made easier by the use of Prolog (see section 6.3). Others that were more difficult (a change in the generator for instance) led to extending the metaprogramming approach to new parts of the software.

8.2.8 Handicaps

Piling up on Prolog and Pascal implies that Véda is dependent on their shortcomings. Pascal compilers rarely implement correctly the whole of ISO Pascal... So we had to adapt to each compiler: and Véda is not completely portable, and the code produced is not either. Those drawbacks of building on high-level environments must be balanced with the gains in building the tool.

9 Conclusion

Lessons have been drawn in section 8. Experience with Véda has shown that this kind of tool corresponds to a real need currently.

Among similar experiences, we can quote SARA [Estrin 86] which provides a lot of tools to analyse specifications but uses a more restricted formalism based on Petri nets.

We are currently doing research work on extensions to Véda in the following directions :

- Performance evaluation (see 5.5).
- Intelligent driving, with backtrack, and strategies that could be based on what is observed; semi-exhaustive simulation.
- Some high-level logic for expressing properties, with automatic translation into an observer (see [Groz 86]).
- Producing code for parallel machines or distributed systems.

Without all these functions though, Véda proved that a tool based on simple simulation of protocols could be useful to a large community of users, as long as it provides a comfortable environment.

10 References

[Barberye 83] G. Barberye, T. Joubert, M. Martin,
Manuel d'utilisation du Prolog-CNET, Technical Report NT/PAA/CLC/LSC/1058, CNET, France, Sept. 1983.

[Bochmann 78] G.v. Bochmann,
Finite State Description of Communication Protocols, Computer Networks, vol. 2, Oct. 1978, pp 361-372.

[CCITT 84] CCITT Red book,
recommendations Z.100 to Z.104.

[Colmerauer 75] A. Colmerauer,
Les grammaires de metamorphoses, Technical Report, GIA, Univ. Marseille-Luminy, France, Nov. 1975.

[Colmerauer 85] A. Colmerauer,
Prolog in 10 Figures, CACM, Vol 28, Nu 12, Dec. 1985, pp 1296-1310.

[Courtiat 87] JP. Courtiat, P. Dembinski, R. Groz, C. Jard,
Estelle, un langage ISO pour les algorithmes distribues et protocoles, TSI, March 1987, vol. 6, nu. 2,pp 89-102.

[Dssouli 86] R. Dssouli, G. V. Bochmann,
Conformance testing with multiple observers, VI IFIP WG6.1 workshop, Gray Rocks, Montreal, June 1986, North-Holland, G. V. Bochmann and B. Sarikaya ed.

[Estrin 86] G. Estrin, RS. Fenchel, R. Razouk, MK. Vernon,
SARA (System ARchitects Apprentice): Modeling, Analysis and Simulation Support for Design of Concurrent Systems, IEEE trans. on SE, Vol 12, Nu 2, Feb. 1986, pp 293-311.

[Groz 85] R. Groz, C. Jard, C. Lassudrie,
Attacking a Complex Distributed Algorithm from Different Sides: an Experience with Complementary Validation Tools, Computer Networks, Vol 10, Nu 5, Dec. 1985, pp 245-257.

[Groz 86] R. Groz,
Unrestricted Verification of Protocol Properties on a Simulation using an Observer Approach, VI IFIP WG6.1 workshop, Gray Rocks, Montreal, June 1986, North-Holland, G. V. Bochmann and B. Sarikaya ed.

[Hoare 78] C.A.R Hoare,
Communicating Sequential Processes, CACM, Vol. 21, Aug. 1978, pp. 666-677.

[ISO 86a] ISO/TC97/SC21/WG16-1 DP9074,
Estelle: a Formal Description Technique based on an Extended State Transition Model, Sept. 1986.

[ISO 86b] ISO/TC97/SC21/WG16-1 DP8807,
Lotos: a Formal Description Technique, Sept. 1986.

[Jard 84] C. Jard,
Protocoles et Services: Test des Specifications, PhD Thesis, Univ. de Rennes, France, Mai 1984.

[**Jard 85a**] C. Jard, JF. Monin, R. Groz,
Experience in implementing X250 in Véda, V IFIP WG6.1 workshop, Moissac, June 1985, France, North-Holland, M. Diaz ed.

[**Jard 85b**] C. Jard, R. Groz, JF. Monin,
Véda: a Software Simulator for the Validation of Protocol Specifications, COMNET'85, Budapest, Oct. 1985, published by North-Holland in *Computer network usage: recent experiences* , L. Csaba, K. Tarnay, T. Szentivanyi ed.

[**Levy 86**] LS. Levy,
A Meta-programming Method and its Economic Justification, IEEE trans. on SE, Vol 12, Nu 2, Feb. 1986, pp 272-277.

[**Linn 85**] RJ. Linn,
The Features and Facilities of Estelle: a Formal Description Technique based upon an Extended Finite State Machine Model, V IFIP WG6.1 workshop, Moissac, France, June 1985, North-Holland, M. Diaz ed.

[**May 83**] D. May,
OCCAM, SIGPLAN notices, vol. 13, nu 4, April 1983, pp 69-79.

[**Monin 84**] JF. Monin,
Ecriture d'un compilateur reel en Prolog, Journees sur la programmation en logique, Plestin, France, April 1984, Cnet ed.

[**Parnas 86**] DL. Parnas, PC. Clements,
A Rational Design Process: How and Why to Fake it?, IEEE trans. on SE, Vol 12, Nu 2, Feb. 1986, pp 251-257.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique